ASPIRE: Automated Systematic Protocol Implementation Robustness Evaluation

Arunchandar Vasan, Atif M. Memon
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: {arun, atif}@cs.umd.edu

Abstract

Network protocol implementations are susceptible to problems caused by their lack of ability to handle invalid inputs. We present ASPIRE: Automated Systematic Protocol Implementation Robustness Evaluation, an automated approach to pro-actively test protocol implementations by observing their responses to faulty Protocol Data Units (PDUs) or messages. In contrast to existing approaches, we sample the faulty PDU space in a systematic manner, thus allowing us to evaluate protocol implementations in the face of a wider variety of faulty PDUs. We use a pruning strategy to reduce, from exponential, the size of the faulty PDU set to polynomial in the number of fields of a PDU. We have implemented the ASPIRE algorithms and evaluated them on implementations of HTTP (Apache, Google Web Server (GWS), and Microsoft IIS) and SMTP (Sendmail and Microsoft Exchange) protocols. Our results show that Apache, GWS, and IIS, although implementing the same protocol specification, behave differently on faulty HTTP PDUs; Sendmail and Exchange are different in handling our faulty SMTP PDUs.

Keywords: automated testing, robustness testing, faulty PDUs, protocol data units, stateless protocols, stateful protocols.

0 Prologue

RedHat Alert. Sep. 30 2003: RedHat[1] reports two bugs in OpenSSL, a popular security protocol implementation, versions 0.9.6 and 0.9.7. The parsing of *unusual ASN.1 tag values* can cause an OpenSSL server to crash. A client can send a certificate with faults in its fields to trigger this bug. The crash can potentially be exploited to launch attacks that may result in controlling the machine remotely.

[1]https://rhn.redhat.com/errata/RHSA-2003-292.html

Microsoft Alert. Oct. 15, 2003: Microsoft[2] reports a vulnerability in the protocol used for its messenger service. A specific *faulty packet* could let an attacker gain control of the system with privileges of the user of the messenger service.

Many problems, similar to the ones outlined above [21] surface frequently in specific protocol implementations. They are usually caused by faulty input, the handling of which is usually left unspecified by the protocol specification. This paper describes techniques to test protocol implementations on such faulty input in an *automated manner*.

1 Introduction

A *protocol* is an *a priori* understanding between two or more hosts (i.e., machines on a network) to communicate by exchanging messages over a network. The messages exchanged are called Protocol Data Units (PDUs) [32, 36]. A *protocol specification* defines the syntax, semantics, and allowed sequences of PDUs. A *protocol implementation* is a set of programs that execute on different hosts achieving the message exchange sequence allowed by the protocol specification.

With the advent and evolution of the Internet, a number of protocols have been specified and implemented. A popular example is the Hyper Text Transfer Protocol (HTTP) [13] protocol usually used to download content from a website. A client sends a PDU with a `GET` message that causes the server to send a file back. Implementations of the HTTP protocol include Apache [1], Google Web Server (GWS) [3], and Microsoft Internet Information Services (IIS) [6].

"Be conservative in what you send, and liberal in what you accept from others" [33]. This design philosophy causes implementations to have different approaches to

[2]http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms03-043.asp

dealing with faulty PDUs. Protocol specifications are typically written in natural language (e.g., English) and often incomplete, especially so, in handling faulty inputs, and many decisions are left to the implementor [23]. As a result, protocol implementations behave differently on faulty inputs, depending on the design decisions made by the implementer.

Many issues in server availability, performance, or even security of a host, come to light as a result of a hacker figuring out that an implementation does not handle some faulty PDU gracefully. Typically, these defects are then fixed by the manufacturer or the open-source community after the break-in has been reported [21].

We advocate pro-active automated testing of protocol implementations on faulty input before deployment on the Internet. This gives greater confidence in the ability of networked systems to survive in the face of malformed input. In this paper, we develop a new technique for *automated testing* of protocol *implementation robustness*. The IEEE defines robustness as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [9]. We define protocol implementation robustness[3] as the *ability of an implementation to handle exceptional input in the form of faulty PDUs and continue normal protocol operation.* An example of abnormal operation would be the premature termination of the process that implements the protocol.

The main challenge in automated robustness testing of protocols is the systematic generation of faulty PDUs. An enumeration of all possible faults is exponential in the number of fields of the PDUs (explained in Section 5). Existing approaches to robustness testing typically assume fault models [17], generate faulty PDUs in an ad-hoc manner, sample all-possible faulty PDUs pseudo-randomly [18], or deal with testing the protocol itself with sequences with fault causing network conditions (e.g., loss of data packets) in a simulation [20] rather than test an implementation with faulty PDUs. We need a method to prune the number of possible faulty PDUs without losing the possible robustness issues revealed by the test cases.

We present a novel approach to automatically test protocol implementations for robustness by generating faulty PDUs in a systematic manner. While our approach is generic enough to be applied to all protocols, we focus on application layer protocols (most protocols which a human user would encounter, e.g., HTTP, Simple Mail Transfer Protocol (SMTP) [24], Domain Name Service (DNS) [31]) as they typically handle human user level input that can be faulty (as opposed to low-level protocols that get "cleaned" input from higher layer protocols). We classify application layer protocols as stateless or stateful. We generate faulty PDUs depending on the protocol's classification. For stateless protocols, we generate syntactically faulty PDUs. As a typical PDU could have many possible syntactic faults (due to many combinations of individual faults in the fields of the PDU) an enumeration would cause an exponential growth in the number of possible syntactically faulty PDUs. We adapt the *pairwise* testing strategy outlined in [35] to prune this growth. For stateful protocols, we use a simple combinatorial enumeration to generate semantically faulty PDUs, where the syntax of the PDU is correct, but the semantics are incorrect, i.e., the protocol does not expect the PDU in that state. Intuitively, we sample the set of all possible faulty PDUs for a protocol systematically and use this sampled set as input for protocol implementations to evaluate their robustness.

We have implemented our technique and empirically evaluated it by creating suites of faulty PDUs for HTTP 1.0, a stateless protocol, and SMTP, a stateful protocol. We are able to reduce the number of faulty PDUs without compromising the evaluation of robustness of an implementation. Our experiments reveal that different implementations of the same protocol (Apache, Google Web Server, Microsoft IIS for HTTP and Sendmail and Microsoft Exchange for SMTP), in fact, behave differently in the face of faulty PDU inputs.

The specific contributions of our work include:

- We propose a taxonomy of application layer protocols on the basis of state information maintained during the protocol execution. We use this taxonomy to design test cases for testing robustness of protocol implementations.
- We show how to compose simple faults in different parts of a PDU to generate faulty PDUs by adapting the pairwise testing strategy described in [35] to robustness testing. Specifically, we generate a set of faulty PDUs such that any pair of faults is present in at least one PDU in this set.
- We show that the reduction in the number of syntactically faulty PDUs to be used for testing by our algorithm does not reduce the capacity to evaluate the robustness of an implementation.
- Finally, we use our algorithm to generate faulty PDUs as test cases for practical protocols, namely HTTP and SMTP, and experimentally evaluate the differences in robustness of different implementations.

In the next section, we discuss related work. In Section 3, we formally define PDUs. We present our taxonomy of application-layer protocols in Section 4. In Section 5, we elaborate on syntactic and semantic faults in PDUs and describe algorithms to systematically generate faulty PDUs. We describe our implementation and empirical evaluation of ASPIRE for HTTP and SMTP in Section 6, and conclude with a discussion of future work in Section 7.

[3]In the rest of this paper, we use robustness and implementation robustness interchangeably.

2 Related Work

Fault injection, i.e., using faulty inputs in test cases, is a well known approach in software testing [37]. Existing work in fault injection for testing protocol implementations needs significant human input in determining failure models and test cases. In [17], the design of a script-driven interface for protocol fault injection is described. The script and failure specification are left to the tester. In [22], *anomalies* are inserted into the protocol specification itself, and used to generate test cases. However, it does not address how many anomalies can exist within one PDU, or how one can systematically generate a number of anomalous PDUs. In [18], fault generation in protocol inputs is chosen in a pseudo-random sampling of the possible exponential number of faulty PDUs.

Studies [20] and [12] address testing protocol robustness, i.e., check if a protocol can handle test sequences with bad network conditions. Note, however, that this is quite different from protocol implementation robustness, i.e., testing implementations with invalid input.

Another related technique is formal protocol testing, which is a well studied problem [34, 25]. In formal conformance testing, the entity executing the protocol is specified using FSMs [19] (Finite State Machine) or a description language such as ESTELLE, LOTOS [14], PROMELA, or SDL [15] which can generate automata on the fly from a specification in the language. The implementation is also represented as an FSM. The problem is now reduced to seeing if the specification and the implementation generate the same output sequence given some input sequence. The key ideas lie in generating input sequences, which, when executed on both, are good enough to say that the implementation conforms to the specification [30]. Other formal approaches [10, 11] view protocol testing as a case of testing concurrent distributed systems. While formal approaches to testing can be applied to test protocol implementation correctness, they do not deal with the robustness of an implementation.

3 PDUs & Protocol Specifications

Intuitively, a PDU consists of data and control information which tell the protocol how to operate on that PDU.

Definition: A *PDU* p is an n-tuple given by $< c_1, \ldots, c_n >$ where each c_i consists of one more bits. The c_is are the control or data fields of a PDU. The number of fields n varies with PDU even within the same protocol. Control fields typically precede the data fields although they can be intermingled within a PDU. □

For example, a HTTP request PDU p_1 is as follows:

$$\boxed{\text{GET}\sqcup\text{url}\sqcup\text{HTTP/version}\backslash n H_1 \backslash n \ldots H_p \backslash n \backslash n}$$

A HTTP response PDU p_2 is of the form:

$$\boxed{\text{HTTP/1.0}\sqcup 200 \sqcup\text{OK}\backslash n H_1 \backslash n \ldots H_q \backslash n DATA \backslash n \backslash n}$$

The symbol $\sqcup$ denotes a space, $\backslash n$ is the newline character, and H_i denotes a header. The GET and the headers are control fields and the url is a data field.

Definition: A *protocol specification* P is a set of sequences S_is, $P = \{S_i | S_i = [< p_1, h_1, a_1 >, \ldots < p_n, h_n, a_n >]\}$ where each triple $< p_i, h_i, a_i >$ represents an action a_i executed on the PDU p_i by the host h_i. The action a_i is either a send or a receive of PDU p_i. □

For the example HTTP PDUs shown, an acceptable sequence of execution is $[< p_1, A, s >, < p_1, B, r >, < p_2, B, s >, < p_2, A, r >]$ where A and B are the communicating hosts, r denotes a receive action, and s denotes a send action. In the case of p_1 the only data field is the url; everything else is a control field. In p_2, the data field appears at the end of the PDU.

4 A Taxonomy of Application Layer Protocols

We use the state information about a client (an entity that *imports* a resource from a network location), maintained by the server (an entity that *exports*, i.e., makes available a resource at a particular location on the network; e.g., a web server), (or peer[4]) in a protocol to classify the application protocol as being *stateless* or *stateful*. By state information about a client, we mean the record of the transaction with the client. For instance, in HTTP/1.0, no such information is maintained about a client. Whereas, in SMTP, a mail server has to keep track of the sequence of commands issued so far by the client.

We now present a taxonomy of application layer protocols. The taxonomy specifies the type of faulty PDUs an application layer protocol might encounter. A scheme based on the classification is used for faulty PDU generation.

Stateless Protocols: When a server does not maintain any state of a transaction, we term it stateless. HTTP 1.0[5] is an example of a stateless protocol. Web browsers and servers implement this protocol. In HTTP 1.0, the server responds to the client when it gets a request with the appropriate file

[4]Peers act as both servers as well as clients in peer-to-peer networks, e.g., in `gnutella [2], morpheus [7], kazaa[4]`, etc.

[5]Web servers typically use cookies to maintain state in an HTTP session. These are not part of the basic protocol.

or error message, closes the connection without maintaining any state information of the client. Note that this specification allows only sequences of length two, i.e., a request followed by a response.

Stateful Protocols: In these protocols, the server maintains state about the client until the entire transaction is over, is aborted, or times out. An example is SMTP, in which a server expects a PDU (e.g., DATA) only after another (e.g., RCPT), thereby maintaining state information about the client.

Note that the state is independent of whether the underlying protocols (such as TCP), which the application level protocol uses, are stateless or stateful. Specifically, although an HTTP server uses TCP, a protocol with states, we classify HTTP as stateless. This is because HTTP server does not keep track of the HTTP client, although its TCP layer tracks the TCP layer of the client.

5 Faults in PDUs

Intuitively, a PDU is faulty if either its syntax is incorrect or its position in a PDU sequence is incorrect. We call the former syntactically faulty PDUs and the latter semantically faulty PDUs. Note that, in general faults of both kinds are also possible in the same PDU.

Definition: A PDU $P =< f_1, f_2, \ldots, f_n >$ is *syntactically faulty*, iff, $\exists i$ such that f_i is faulty, i.e., outside the range of the inputs the protocol accepts as valid. A PDU P is maximally faulty if $\forall i$ f_i is faulty. □

For a systematic evaluation of robustness, we need to consider the execution of a protocol implementation with all combinations of faults and valid values in its fields, which are too many in number. We observed by looking at implementations that it suffices to test for maximally faulty PDUs. However, this is a heuristic. Our future work will look at the effect of choosing maximally faulty PDUs on our results. Let $E_{f_i} = \{e_1, e_2, \ldots e_m\}$ denote the set of faults for a particular field f_i. The size of the set of maximally faulty PDUs is m^n if each of the n fields has m possible faults. Therefore, even the number of maximally faulty PDUs grows exponentially in the number of fields. This is a problem and any testing technique must test a subset.

Note that if a protocol is stateless, it is susceptible to syntactically faulty PDUs. An incorrect sequence of syntactically correct PDUs cannot cause problems for a stateless protocol since it maintains no history of PDUs. However, if a protocol is stateful, it is susceptible to syntactically and semantically faulty PDUs.

Definition: Let $S = \{s_1, s_2, \ldots, s_n\}$, be the set of states of a stateful protocol. Let $A(s_i)$ be the set of syntactically correct PDUs that are *acceptable* in state s_i. Then a PDU P is semantically faulty if $P \notin A(s_i)$. □

Intuitively, the protocol does not expect its reception in state s_i, and so it is semantically faulty.

Let there be m states in a stateful protocol, and n possible valid PDUs for that instance of protocol execution (i.e., for some valid chosen values of PDU fields). Then, the number of semantically incorrect PDUs is at most mn, which is well manageable. This is because for each state there are at most n invalid PDUs (as the maximum number of possible PDUs is clearly more than the number of PDUs unaccepatable in that state). Therefore, the number of semantic faults grows polynomially for any instance of protocol execution with the number of states in the protocol as well as the number of types of PDUs in the protocol.

5.1 Generating Syntactically Faulty PDUs

The key idea in generating syntactically faulty PDUs systematically is that the tester seeds possible faults for each field of the PDU, which are then combined by our algorithm for faulty PDU generation. To prune the number of possible faulty PDUs and prevent exponential growth, we adapt a well-known result from software testing called the pair-wise testing constraint [16, 35]. The algorithm is the same as the one proposed in [35], but for our usage in the context of testing protocol implementations. We impose the constraint that *the set of faulty PDUs is chosen such that each pair of faults is satisfied by at least one PDU*. This constraint has been shown to provide effective coverage for predicate testing.

Suppose field A of a PDU has the possible faults a_1, a_2, field B has the possible faults b_1, b_2, and field C has the possible faults c_1, c_2, c_3. Then, the set of triples $S = \{< a_1, b_1, c_1 >, < a_1, b_2, c_2 >, < a_2, b_1, c_3 >, < a_2, b_2, c_1 >, < a_2, b_1, c_2 >, < a_1, b_2, c_3 >\}$ satisfies the property that each pair of faults appears in at least one of the triples. Note that a simple combinatorial enumeration of the faulty PDUs has the size 2 × 2 × 3 = 12, whereas the pairwise strategy yields a set that has 6 members in it. Also, note that there are many such sets which satisfy the constraint imposed; we need to generate one such set which satisfies the constraint.

Algorithm 1 outlines our approach for generating syntactic faults. The algorithm has two phases, horizontal and vertical growth. Intuitively, the horizontal growth first enumerates all possible pairs of errors for the first two fields. For the remaining fields, *it adds its possible faults by augmenting the existing set of PDUs such that the choice minimizes the number of pairs left uncovered*. The set size does not grow the set of PDUs beyond the $O(n^2)$ pairs generated by enumerating all possible pairs for the first two fields considered.

The vertical growth, on the other hand, *adds a new PDU for a pair with other fields of the PDU set to a don't care value (D), i.e., any value can be used later to fill this po-*

Algorithm 1: SYNTACTIC-FAULT-GENERATION( )

```
1  $P \leftarrow$ all pairs of faults in each pair of PDU fields
2   /* Phase 1 of Horizontal Growth */
3  for each $f_1$ in set of faults for field $p_1$
4  do for each $f_2$ in set of faults for field $p_2$
5      do $S_i \leftarrow < f_1, f_2 >$
6         $i \leftarrow i + 1$
7   /* Phase 2 of Horizontal Growth */
8  for each $f$ in set of fields
9  do for each $f_i$ in set of faults of $f$
10     do $S_i \leftarrow < S_i : f_i >$
11        Remove all pairs satisfied from $P$
12     for each $S_i$ in remaining entries
13     do Choose $f_j$ that satisfies maximum
14        number of pairs in $P$
15        $S_i \leftarrow < S_i, f_j >$
16        Remove all satisfied pairs from $P$
17  /* Vertical Growth */
18 $S' \leftarrow \emptyset$
19  /* $i < k$ is imposed to avoid counting pairs */
20  /* twice. $w_i$ is the fault in field $p_i$ and */
21  /* $u_k$ is a fault in field $p_k$ */
22  /* "D" stands for a don't care value */
23 for each unsatisfied pair $< w_i, u_k >, i < k$ in $P$
24 do if $\exists s \in S'$ such that $p_k = D$ and fault $p_i = w_i$
25       then replace the D with $w_k$
26       else add a faulty PDU with $p_i = w_i$
27            and $p_k = u_k$ and $\forall_{j \neq i,k}\{p_j = D\}$
28             Remove newly satisfied pairs from $S'$.
29 $S \leftarrow S \cup S'$
```

sition by the algorithm. For every pair, the algorithm first checks if the pair can be satisfied within an existing PDU one of whose values is D and the other's value is the same as in the partial PDU. If so, the D field in the partial PDU is replaced with the new value. Otherwise, it constructs a new partial PDU with the values of two fields set from this pair and other fields set to D.

We now trace through the algorithm using two parameters A and B. The only set that satisfies the pairwise-constraint is $F(A, B) = \{< a_1, b_1 >, < a_1, b_2 >, < a_2, b_1 >, < a_2, b_2 >\}$, which is the same as the combinatorial enumeration.

Now consider the case of three parameters A, B, and C. We extend $\{< a_1, b_1 >, < a_1, b_2 >, < a_2, b_1 >\}$ by the parameters c_1, c_2, c_3 to get $\{< a_1, b_1, c_1 >, < a_1, b_2, c_2 >, < a_2, b_1, c_3 >\}$. This is the first phase of the horizontal growth algorithm as shown in lines 1-6 where the original set $S(A, B)$ is augmented with faults from each of the remaining fields.

Now this leaves the pairs $\{< a_2, c_1 >, < b_2, c_1 >, < a_2, c_2 >, < b_1, c_2 >, < a_1, c_3 >, < b_2, c_3 >\}$ unsatisfied. The next phase of the horizontal growth algorithm in lines 8-16 augments all those rows of the existing set $S(A, B)$ by that fault of the new field, that minimizes the number of uncovered pairs. In this case, the choice of c_1 to $< a_2, b_2 >$ covers two missing pairs $< a_2, c_1 >$ and $< b_2, c_1 >$. This is the maximum number of pairs that can be covered. This process is repeated for each remaining field in the PDU.

Note that the number of faulty PDUs that are generated by this horizontal growth never exceeds $|A| * |B|$, the cardinality of the set $S(A, B)$. At the end of this horizontal growth, some pairs are still unsatisfied. Therefore, the PDU generation algorithm enters the next part called the vertical growth in lines 18-24, where the number of faulty PDUs is increased.

For the pair of faults $< a_2, c_2 >$, we have the new faulty PDU $< a_2, D, c_2 >$, where D stands for a dummy don't care value. Likewise, we have to add $< a_1, D, c_3 >$ to cover $< a_1, c_3 >$. Now, to cover $< b_1, c_2 >$, we can simply change $< a_2,, c_2 >$ to $< a_2, b_1, c_2 >$ and to cover $< b_2, c_3 >$ we change $< a_1, D, c_3 >$ to $< a1, b_2, c_3 >$ which covers all pairs of faults.

It can be shown that, on termination, the algorithm outputs a set of faulty PDUs with the property that any pair of faults in two fields is satisfied by at least one PDU in this set. Furthermore, the size of this set does not grow exponentially. A detailed analysis of the algorithm is beyond the scope of this paper. The interested reader is referred to [35] for details.

Algorithm 2: SEMANTIC-FAULT-GENERATION( )

```
1 $S \leftarrow \Phi$
2 for each $s$ in set of states of the protocol
3 do for each PDU $p$ in protocol instance
4     do if $p \notin A(s)$
5          then $S \leftarrow S \cup < s, p >$
6 for each $s$ in set of states of protocol
7 do Compute $R(s)$, the sequence
8      of PDUs from start state which leads to $s$.
```

5.2 Generating Semantically Faulty PDUs

As the number of semantically faulty PDUs does not grow exponentially with the number of states and PDUs, we do not worry about reducing the number of semantically faulty PDUs. Consequently, the algorithm we use is a straightforward enumeration of the number of semantically faulty PDUs for each state. We present our approach in Algorithm 2.

Each state in a stateful protocol can accept only certain PDUs. Therefore, for each state, the remaining PDUs of the

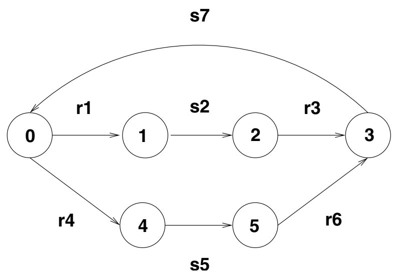

Figure 1. A Stateful Protocol Specification

protocol are semantically faulty. The algorithm *enumerates the semantically faulty PDUs* for each state. When testing, however, one should first drive the protocol implementation to a particular state using correct PDUs before injecting a semantically faulty PDU in that state to be tested. Therefore, the test case consists of a state, a list of semantically faulty PDUs, associated with that state, and a sequence of correct PDUs which will drive the protocol to that state. This is similar to our earlier work [26, 27, 28, 29].

We illustrate our algorithm with a simple stateful protocol specified in Figure 1 as a directed graph. The starting state of the protocol is 0. An arc labeled r_i means that the machine receives a PDU i. Likewise, an arc labeled s_i stands for sending a PDU i. Consider state 5, it expects to receive PDU 6. Now, the algorithm would generate $\{< 5, 1 >, < 5, 3 >, < 5, 4 >\}$ for state 5, i.e. PDUs 1, 3, and 4 are semantically faulty in state 5. And, the path for driving the implementation would be r_4, s_5, i.e., the testing program would send the PDU 4 and receive the PDU 5, and then send the semantically faulty PDU.

6 Experiments

We now show the practicality and usefulness of our technique. Specifically, we experimentally evaluate the following hypotheses:

- Different protocol implementations offer different robustness to faulty PDU input, and this is true for both stateless and stateful protocols.
- The pair-wise faults constraint does not reduce the capacity to evaluate robustness of different implementations.

6.1 Metrics for Evaluation

We define the following metrics for studying the performance of our algorithm and test-case generation.

- *The Reduction Ratio* is the ratio of the number of faulty PDUs generated by the syntactic faulty PDU generation algorithm to combinatorial enumeration.
- *The Robustness Ratio* is the ratio of the number of faulty inputs that were handled by an implementation by continuing normal operation to the total number of faulty inputs.

To ensure our hypotheses hold, we show that:

- the average robustness ratio as estimated with our syntactic faulty PDU set is about the same as that of the set of all possible faulty PDUs.
- the average robustness ratio of different implementations is different for both syntactic and semantic faults.

We use the reduction ratio to study the extent of reduction in the number of test-cases needed. For studying the performance of semantic faulty PDUs, we itemize the effect of each of the PDU and its faulty state.

6.2 Threats to Internal Validity

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher's knowledge. The *independent variables* in our experiments are the protocol implementations under test and the test suites generated by our algorithms. The *dependent variables* are the robustness ratios of the implementations. The syntactic mutants chosen by us for the initial seeding of faults in different fields is the cause for greatest concern. Our approach was to limit this by logically splitting the domain of faulty inputs of each field of the PDU and choosing representatives from each of these sub-domains.

6.3 Threats to External Validity

Threats to External Validity are conditions that limit the ability to generalize the results of our experiments to industrial practice. We consider artifact representativeness to be the major threat to external validity. The subject programs themselves are widely used products, and therefore are representative of industrial practice. However, content is dynamically generated. Our experiments don't represent dynamic URL creation. However, we can generate URLs that represent actual faulty input once we study common URL generation patterns used in practice.

6.4 Threats to Construct Validity

Threats to construct validity are conditions caused by the inadequacy of the measuring metrics in capturing the property measured. We have defined the robustness ratio as the number of faulty test cases resulting handled properly by the implementation divided by the total number of faulty PDUs sent as inputs. While this does indicate the degree of robustness, this is a fairly coarse metric and may not factor the cost of processing each PDU.

6.5 Subject Protocols

We chose HTTP and SMTP as subject protocols for our study. HTTP is a simple stateless protocol, and SMTP is a simple stateful protocol. We focus on the most commonly used PDUs of these protocols. We implemented our syntactic fault generation algorithm for HTTP and semantic fault generation algorithm for SMTP.

6.5.1 The HTTP Protocol

The core operation of HTTP is to fetch content using the GET command. The GET PDU has several headers, and the typical operation is as follows:

- Client sends GET PDU
- Server sends content.

Our approach generates syntactically faulty request PDUs with syntactic mutants of the GET command.

6.5.2 The SMTP Protocol

The SMTP protocol is executed by mail servers of a domain and is used to exchange electronic mail between different domains (e.g. mailin-02.mx.aol.com is the mail server for the domain aol.com.) Let C denote a client and S denote a server. The notation $A \rightarrow B$ P means that A sends PDU P to B. SMTP works as follows:

- $S \rightarrow C$ 220 mail-server-name SMTP
- $C \rightarrow S$ HELO domainname
- $S \rightarrow C$ 220 Hello domainname pleased to meet you
- $C \rightarrow S$ MAIL FROM:
- $S \rightarrow C$ 250 Sender OK
- $C \rightarrow S$ RCPT TO:
- $S \rightarrow C$ 250 Recipient OK
- $C \rightarrow S$ DATA
- $S \rightarrow C$ 354 Enter mail, end with a "." on a line by itself
- $C \rightarrow S$ Message Content
- $S \rightarrow$C 250 Message queued for delivery
- $C \rightarrow$S QUIT
- $S \rightarrow$C mail-server-name closing connection

The protocol has 4 states, each of which it reaches from the start state for the PDUs HELO, MAIL, RCPT, DATA, and QUIT respectively. As an illustration, sending a DATA before MAIL and RCPT is a semantic fault. We test SMTP server implementations, by issuing semantically faulty PDUs generated by our algorithm.

6.6 Results of HTTP Test Suite

We tested the web servers www.umd.edu that runs Apache 2.0.45, www.google.com that runs GWS 2.0 (Google Web Server), and www.microsoft.com that runs IIS 6.0 (Internet Information Services). The test PDUs were generated from the simple faults being compounded to maximally faulty HTTP GET requests by our algorithm for syntactic faults. We automated it by a driver program, which sends each of the requests to the web server over the Internet, and logs the response. The driver program first resolves the name of the web server to be tested to get the IP address and sets up a TCP connection to port 80 on the server using the BSD socket library. Then, the program reads an entry from the precomputed test suite and sends a maximally faulty request to the server. Finally, it reads the response from the server if any. For each PDU, the driver program sets up a new connection and repeats the entire process.

The GET PDU is the most commonly used HTTP PDU. For instance, publicly available statistics of the www.cs.umd.edu web-server say that out of the 1,121,178 requests over a period of one week, 1,106,119 (98.6%) were GET requests. Therefore, we focus on the GET PDU for our HTTP testing. We chose 5 fields namely url, version, date, if-modified-since, and refer for the GET PDU. The url used as the base for faulty URLs was "/" indicating that the main page of the webserver is requested. These were seeded with 10 syntactic faults each and combined according to the algorithm described before to produce a syntactically faulty PDU set. As a result of the pruning algorithm we had 801 cases, instead of the possible 10^5. Table 1 summarizes the results of our pruned syntactically faulty PDU test suite for HTTP. Each row of the table gives the number of requests that were considered "Normal" and the number of those that resulted in "Exceptional" behaviour of the server. By exceptional, we mean that the server process possibly crashed[6] or the TCP connection was terminated improperly. We say "possibly" crashed because we do not have access to the error/crash logs of any of the servers themselves, and are inferring this from our observed response or lack of it. In our future work, we will evaluate our testing approach with servers we control.

Figure 2 shows the robustness ratios observed for different servers. The X-axis shows the size of the set of faulty PDUs used to test the system and the Y-axis shows the robustness ratio observed. The curve for Microsoft IIS stays flat at zero showing lack of robustness in handling faulty PDUs. The GWS curve shows that it handles some of the smaller-sized sets well but the overall robustness ratio falls off to a very small non-zero value with increasing number of faulty PDUs. This is due to newer combinations of faults which were not present in the smaller-sized sets. Apache shows the best behaviour. Although the robustness ratio os-

[6] Typically, a web server consists of several processes executing the HTTP protocol. Even if one process crashes, the others can still handle requests.

Server	"Exceptional" responses	"Normal" responses
Apache	19	782
GWS	792	9
IIS	801	0

Table 1. Responses to faulty HTTP PDUs of Pruned Suite

Server	Pruned	All-Cases
Apache	0.9762	0.97
GWS	0.0112	0.02
IIS	0.0	0.0

Table 2. Overall Robustness Ratios

cillates for small-sized sets showing variations in its ability to handle combinations of faults, it converges to a value very close to 1. The overall values of robustness ratios are shown in Table 2.

The results conclusively show that different implementations react to faulty PDUs differently. More interestingly, some of the "Normal operation" responses were those of HTTP/1.1, though the request PDUs were those of HTTP/1.0. This is because most web server implementations implement both versions 1.0 and 1.1 of HTTP. Such responses show that the implementation was confused by the faults to classify it as a different version of the protocol- but handled it normally. In general, according to our definition of robustness, Apache is the most robust, while GWS is more robust than IIS, which is the least robust.

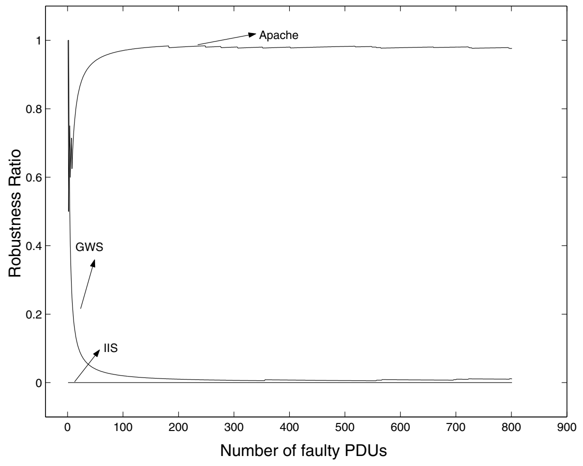

Figure 2. Robustness ratio vs. number of syntactically faulty HTTP PDUs: With pruning

In order to verify that the pruning did not reduce the ability to evaluate robustness, we tested the servers with all 10^5 possible faulty PDUs over several days and the results are shown in Figure 3. The X-axis shows the number of faulty PDUs and the Y-axis shows the robustness ratio. The trends are very similar to to those of the pruned-set. Specifically, the robustness ratios converge to those shown in Table 2. Each row in the table shows the robustness ratio computed for the pruned and the non-pruned all-faulty PDUs test suites. The results show that our approach is effective in pruning without compromising on robustness evaluation.

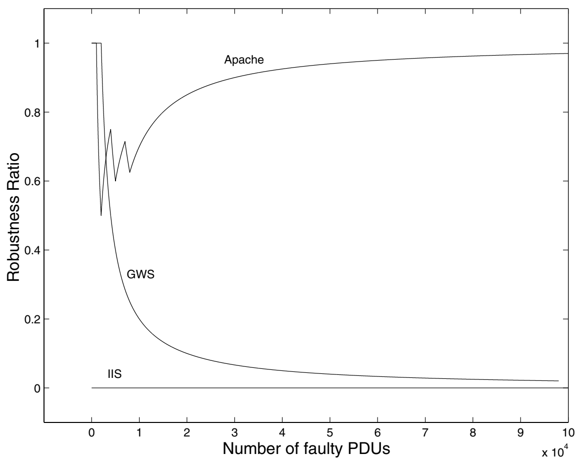

Figure 3. Robustness ratio vs.number of syntactically faulty HTTP PDUs: No pruning

6.7 Results of SMTP Test Suite

We tested the mail-server hosts ringding.cs.umd.edu, that uses the sendmail 8.1.19 [8], an open-source implementation, and envoy.cs.umd.edu that uses Microsoft Exchange 5.5 [5]. We itemize the results of the tests rather than compute the robustness ratio, as the number of tests is small. The results of the tests are shown in Tables 3 and 4. Each row of the tables records the response of an implementation to semantically faulty PDUs, when it expects a PDU shown in column 1. A $\surd$ denotes that the PDU was handled by the server and normal operation proceeded, while a $\times$ denotes it was not so. NA stands for not-applicable, i.e., the PDU is not semantically faulty. The robustness ratios of the two implementations are different, and the sendmail implementation is more robust towards semantically faulty PDUs than the Exchange implementation.

6.8 Evaluation of Pair-wise Fault Generation

Finally, we studied the effectiveness of our pruning strategy. Specifically, we need to confirm that the strategy

Sent	HELO	FROM	RCPT	DATA	QUIT
Expected	Observed Responses				
HELO	NA	√	×	×	√
FROM	√	NA	×	×	√
RCPT	√	×	NA	×	√
DATA	√	×	√	NA	√
QUIT	×	√	×	×	NA

Table 3. Responses of sendmail **to semantically faulty PDUs.**

Sent	HELO	FROM	RCPT	DATA	QUIT
Expected	Observed Responses				
HELO	NA	×	×	×	√
FROM	×	NA	×	×	√
RCPT	√	×	NA	×	√
DATA	×	×	√	NA	√
QUIT	×	√	×	×	NA

Table 4. Responses of Exchange **to semantically faulty PDUs.**

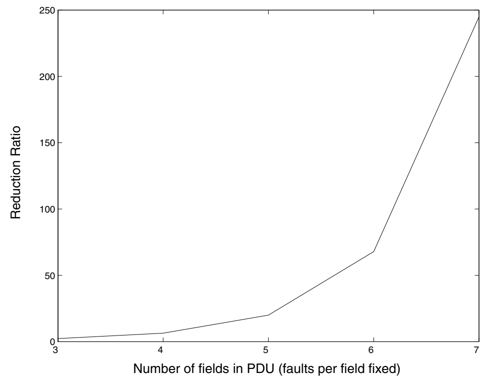

Figure 4. Reduction Ratio vs. number of fields in PDU for faults per field = 5

scales, i.e., achieves greater reduction, with increasing number of fields as well as more number of seeded faults per field. Therefore, we fixed n, the number of fields, and varied the number of faults m for each field. Then, we fixed m, and varied n. The results are shown in Figures 4 and 5 respectively.

In both cases, the Y-axis shows the reduction ratio, i.e., the extent of reduction achieved, and the X-axis shows the parameter being varied. In both cases, the reduction ratio is an increasing function, showing that the extent of reduction is able to keep pace with growth of number of fields in a PDU as well as number of faults per PDU. We have already shown that the effectiveness in evaluating robustness is not compromised by this pruning strategy.

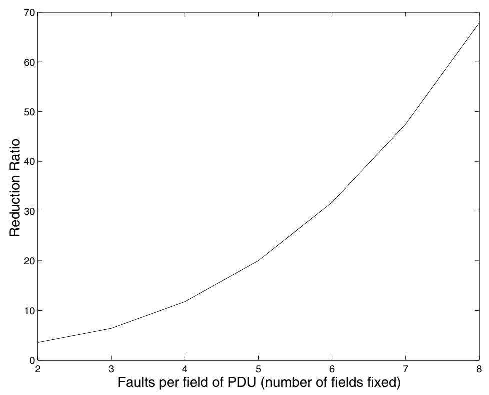

Figure 5. Reduction Ratio vs. faults per field for number of fields = 5

7 Conclusions and Future Work

Network protocol implementations are susceptible to misbehaving clients. Robustness testing of implementations gives greater confidence in deployed systems. Our approach is to pro-actively test protocol implementations by injecting faulty PDUs. A protocol can have syntactically and semantically faulty PDUs depending on whether it is stateless or stateful. As a PDU typically has many fields, the number of possible syntactically faulty PDUs grows exponentially in the number of fields even for a fixed number of faults for each field. We proposed a pruning strategy to reduce the number of such possible faults by imposing an all-pairs constraint, i.e., the set of faulty PDUs is chosen such that each pair of faults between any two fields is satisfied by at least one PDU. For stateful protocols, we defined semantically faulty PDUs that are syntactically correct but are received in a wrong state of the protocol. We used a simple enumeration to generate a set of semantically faulty PDUs.

We implemented our approach for HTTP and SMTP and evaluated its performance empirically. Our test suites consist of syntactically faulty PDUs for HTTP and semantically faulty requests for SMTP. We showed that different protocol implementations have different robustness. We also verified that our pruning sharply reduces the size of the set to be tested without significantly reducing the ability to evaluate robustness.

In the future, we will consider the impact of choosing a non-maximally-faulty set and pruning such a set. We intend to test servers which we control to give us a better idea of what exactly happened to the server process. Finally, our preliminary study of a hybrid testing technique using both syntactically and semantically faulty PDUs for SMTP shows promising results. We would also like to see how code coverage for faulty PDU handling differs from code coverage for normal PDU handling. We will also try semantic fault injection with protocols that have more states such as Yahoo!, MSN, and syntactic faults for dynamically created HTTP URLs.

8 Acknowledgements

We thank Adithya Nagarajan of the Department of Computer Science, University of Maryland and the anonymous referees for their useful comments.

References

[1] Apache Web Server http://www.apache.org.
[2] Gnutella File Sharing Program http://www.gnutella.com.
[3] Google Web Server http://www.google.com.
[4] Kazaa File Sharing Program http://www.kazaa.com.
[5] Microsoft Exchange Server http://www.microsoft.com/exchange.
[6] Microsoft Internet Information Services (IIS) http://www.microsoft.com/iis.
[7] Morpheus File Sharing Program http://www.morpheus.com.
[8] Sendmail Mail Server http://www.sendmail.org.
[9] I. S. 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. Dec. 1990.
[10] A. U. Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3):225–262, Sept. 1993.
[11] A. U. Shankar and S. S. Lam. Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties. *Distributed Computing*, (2) pp. 61-79, 1987.
[12] S. Begum, M. Sharma, A. Helmy, and S. K. S. Gupta. Systematic Testing of Protocol Robustness: Case Studies on Mobile IP and MARS. In *LCN*, pages 369–380, 2000.
[13] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. In *RFC1945, http://www.rfc-editor.org*, 1996.
[14] E. Brinksma, G. Scollo, and G.Steenbergen. LOTOS Specifications, Their Implementations, and Their Tests. *Proceedings of Sixth International Workshop on PSTV*, 1986.
[15] L. Bromstrup and D. Hogrefe. TESDL: Experience with Generating Test Cases from SDL Specifications. *Proceedings of Fourth SDL Forum*, 1989.
[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7), 1997.
[17] S. Dawson and F. Jahanian. Probing and fault injection of protocol implementations. In *International Conference on Distributed Computing Systems*, pages 351–359, 1995.
[18] J. Griffin. Testing Protocol Implementation Robustness. In *29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconson*, 1999.
[19] G.V.Bochmann. Finite State Description of Communication Protocols. *Computer Networks*, 2, 1978.
[20] A. Helmy. Systematic Testing of Multicast Protocol Robustness, 1997.
[21] http://www.cert.org. *The CERT Coordination Center, Software Engineering Institute, CMU*.
[22] http://www.ee.oulu.fi/research/ouspg/protos/. PROTOS-Security Testing of Protocol Implementations.
[23] http://www.rfc editor.org. *Request for Comments*. Specifications of Protocols.
[24] J. Klensin. Simple Mail Transfer Protocol. In *RFC2821, http://www.rfc-editor.org*, 2001.
[25] R. J. Linn and M. U. Uyar. Conformance Testing Methodologies and Architechtures for OSI Protocols. *IEEE Computer Press*, 1994.
[26] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.
[27] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, Nov. 8–10 2000.
[28] A. M. Memon, M. E. Pollack, and M. L. Soffa. Plan generation for GUI testing. In *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 226–235. AAAI Press, Apr. 2000.
[29] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb. 2001.
[30] R. E. Miller and S. Paul. Generating Minimal Length Test Sequences for Conformance Testing of Communication Protocols. *Proceedings of INFOCOM*, 1991.
[31] P. Mockapetris. Domain names - implementation and specification. In *RFC1035, http://www.rfc-editor.org*, 1987.
[32] L. Peterson and B.Davie. *Computer Networks: A Systems Approach*. Morgan Kaufman, 2000.
[33] J. Postel. Transmission Control Protocol. In *RFC793, http://www.rfc-editor.org*, 1982.
[34] B. Sarikaya, G.V.Bochmann, and E.Cerny. A Test Design Methodology for Protocol Testing. *IEEE Transactions on Software Engineering*, 28(1), 2002.
[35] K.-C. Tai and Y. Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28(1), 2002.
[36] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2000.
[37] J. Voas and G. McGraw. *Software Fault Injection: Incoculating Programs Against Errors*. John Wiley and Sons, 1998.